Assignment #4

Authors: *Ryan Ceresani*

Class: *605.744 Information Retrieval*

Term: *Fall 2021*

This assignment submission is structured slightly different than my previous - as we are using a Notebook in place of a "main" application. The notebook provides a nice integration for intermediate code output, plotting, results and experimentation.

Expected Deliverables

- Report outlining methodologies, tools used, parameter decisions, etc.
- Precision, Recall, F1 score for the `"dev"` dataset using "Title" column as features.
- Metrics for `"dev"` dataset using "Title", "Abstract", and "Keywords" as features.
- Perform an additional non-trival experimentation.
- Predictions for the `"test"` dataset.

Results First Reporting

We will dive into all of the details later but will jump straight into the results. Specific details for each run will be visible in the Notebook cells below.

Initial Attempt: SingleLinearLayerModel

- **Data Set Details**
  - Initialized Embedding that was Trained only on the Training Dataset
  - Used only "Title" as feature.
- **Training Details**
  - Epochs = 50
  - Learning Rate = 0.001
- **Results**
  - Precision: 56 / (56 + 348) = **0.1386**
  - Recall: 56 / (56 + 94) = **0.3733**
  - F1: 2 (*0.1386* 0.3733) / (0.1386 + 0.3733) = **0.2021**
  - Avg Precision: 0.07113

Experimentation: GloVeMLP

Because of bad results - did experimentation before moving on to using "Title" "Abstract" "Keywords"

- **Data Set Details**
  - Used only "Title", "Abstract", "Keywords" as features.
  - GloVe 840B.300d used as vocab + initalized embeddings for model.
  - **Training Details**
  - Same as above
  - **Results**
  - Precision: 126 / (126 + 580) = **0.1785**
  - Recall: 126 / (126 + 24) = **0.84**
  - F1: 2 (*0.1785* 0.84) / (0.1785 + 0.84) = **0.2944**
  - AP: **0.15486346777255336**

Tensorboard Outputs

Training Dev

Overall Methodology

Before getting into code specifics, we will address some overall elements used during this assignment. The main approach I am choosing to perform **Text Classification** is through deep learning.

Open Source Libraries

To facility deep learning, this experiment makes heavy use of `PyTorch` and `torchtext` for doing the underlying operations. Additionally, `scikit-learn` is useful for their `metrics` library which offers useful calculation ability for a variety of classifation metrics (whether you use one of their estimators or not.)

- **PyTorch**: This is the heavy-lifter for a number of backing abstract classes performing a variety of functions.
- **Torchtext**: Extension off the official PyTorch to provide text based utilities.
  - `vocab` : This generates a `Vocabulary` object from an iterator which can be used to map words to indices, converting tokens into values.
  - `utils` : It also has some built in utilities for tokenizing and creating ngrams.
- **scikit-learn**: Used for the `metrics` library to generate scores and repots.

Dataset

Systematic Review

- NOTE: The Document **hash:a8113f0b-6561-3178-8c2d-7b4ebac229ff** contained an odd sequence of characters in UTF8 at the beginning of the "Article" section which caused problems for the python stdlib `csv_reader`. The value was sanitized to remove the characters (`",`) - which would be removed in tokenization anyway.

Dataset Challenges

A primary challenge imposed by this dataset is the vast class imbalance within training. (30:1 negative skew)

To counteract this, two approaches considered were:

1. The `DataLoader` used a `WeightedRandomSampler` which was weighted to allow the random sampler topopulate batches with a statistically equal chance for both classes.
2. The chosen loss function (`CrossEntropyLoss`) used inverse class frequency weighting to encourage learning on the minority class.
   - This ended up providing less benefit and the combination of both was worse.

Models

After numerous experimentation, two models were used during the course of this. Both used an `EmbeddingBag` instead of a typical embedding. It provides a lot of functional benefits but does remove the ability to consider sequences of words. So the model loses all context of the sentences themselves.

The actual code for the models will be appended to the end of the notebook.

Simple Linear Model

The first draft model used the embedding bag and a single Linear Layer. The results were not very promising and it was abandoned. (It was tested on the TAK features as well with limited change.)

MLP Model

In an effort to increase the generalization, which was a problem with the first model, I created a straight-forward MLP. The inclusion of activations, in this case, `GELU`, provided for non-linearity and seemed to increase performance.

- 3 fully-connected layers
- GELU Activation
- Dropout during training

Experiment Walkthrough

We will now walk through the experiment notebook to see results in action.

Imports

The open source and custom modules used are imported first.

In [2]:
```
%load_ext autoreload

%autoreload 2
```

In [1]:
```
import torch
from torch.utils.tensorboard import SummaryWriter
from torchtext.data.utils import ngrams_iterator
from torchtext.data.utils import get_tokenizer
```

In [2]:
```
from ir_classification import datasets, models
from ir_classification import vocab as ir_vocab
from ir_classification import train
```

Setup the Training and Validation Datasets

This first time through we do not use GloVe, just the original vocab.

In [3]:
```
datafield_map = {"assessment": 0, "doc_id": 1, "title": 2, "authors": 3, "journal": 4, "issn": 5, "year": 6, "l
data_columns = [datafield_map["title"]]
ngrams = 1
batch_size = 64

# Create vocab from the training data.
vocab = ir_vocab.create_vocab_from_tsv("../datasets/systematic_review/phase1.train.shuf.tsv", data_columns, ngr

# Load the TSV into datasets with the appropriate feature columns.
train_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic_review/phase1.train.shuf.tsv", data_colum
val_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic_review/phase1.dev.shuf.tsv", data_columns)

# Create the transforms for the dataloader to appropriately format the contents of the files.
label_transform = lambda x: x if x > 0 else 0
tokenizer = get_tokenizer("spacy")
text_transform = lambda x: list(ngrams_iterator(tokenizer(x), ngrams))

# Instantiate the dataloaders.
train_dataloader = datasets.create_torch_dataloader(train_dataset, vocab,  label_transform, text_transform, wei
val_dataloader = datasets.create_torch_dataloader(val_dataset, vocab, label_transform, text_transform, weighted
```

```
C:\Users\ryanc\AppData\Local\pypoetry\Cache\virtualenvs\ir-classification-_Pgcz6ju-py3.9\lib\site-packages\torc
htext\data\utils.py:123: UserWarning: Spacy model "en" could not be loaded, trying "en_core_web_sm" instead
  warnings.warn(f'Spacy model "{language}" could not be loaded, trying "{OLD_MODEL_SHORTCUTS[language]}" instea
d')
```

Instantiate the Model

In [4]:
```
num_classes = 2
vocab_size = len(vocab) # from vocab created earlier.
embedding_size = 64
hidden_layer_size = 100

# Enable compatability when training with GPU enabled devices.
# (Some development work was done in Google Colab with GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = models.EmbeddingBagLinearModel(vocab_size, embedding_size, num_classes).to(device)
```

Setup the top-level training loop

The custom code was meant to handle the individual `step` and `epoch` levels generically. This setup should let us change the components experimentally in cells like below without much other hassle.

In [5]:
```
EPOCHS = 50
learning_rate = 0.001

# Create the loss function weighted to inverse class distribution
# loss_function = torch.nn.CrossEntropyLoss(weight=train_dataset.class_weights)
loss_function = torch.nn.CrossEntropyLoss()

# Instantiate a Stochastic Gradient Descent optimizer and "Auto" Learning Rate schedule.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.95)

# Tensorboard writing utility class.
log_dir = "runs/SimpleLinearModel"
writer = SummaryWriter(log_dir=log_dir)

# Perform Training
for i in range(EPOCHS):
    start_iter = len(train_dataloader) * i
    train.train_epoch(i, model, optimizer, loss_function, train_dataloader, start_iter=start_iter, writer=write
    validation_results = train.evaluate_epoch(i, model, loss_function, val_dataloader, writer)
    scheduler.step()


torch.save(model.state_dict(), "model_weights/title_only_SLM_state_dict.pth")
```

```
         Epoch 0: 100%|██████████| 340/340 [00:08<00:00, 40.23 batch/s, accuracy=1, loss=0.459]
    Validation: 0: 100%|██████████| 76/76 [00:00<00:00, 89.42 batch/s, accuracy=0.82, loss=0.413]
         Epoch 1: 100%|██████████| 340/340 [00:07<00:00, 48.20 batch/s, accuracy=1, loss=0.268]
    Validation: 1: 100%|██████████| 76/76 [00:00<00:00, 140.39 batch/s, accuracy=0.92, loss=0.266]
         Epoch 2: 100%|██████████| 340/340 [00:06<00:00, 49.98 batch/s, accuracy=1, loss=0.126]
    Validation: 2: 100%|██████████| 76/76 [00:00<00:00, 127.11 batch/s, accuracy=0.74, loss=0.514]
         Epoch 3: 100%|██████████| 340/340 [00:07<00:00, 47.85 batch/s, accuracy=1, loss=0.0499]
    Validation: 3: 100%|██████████| 76/76 [00:00<00:00, 140.08 batch/s, accuracy=0.86, loss=0.308]
         Epoch 4: 100%|██████████| 340/340 [00:06<00:00, 50.22 batch/s, accuracy=1, loss=0.0969]
    Validation: 4: 100%|██████████| 76/76 [00:00<00:00, 132.20 batch/s, accuracy=0.92, loss=0.355]
         Epoch 5: 100%|██████████| 340/340 [00:06<00:00, 50.21 batch/s, accuracy=1, loss=0.0385]
    Validation: 5: 100%|██████████| 76/76 [00:00<00:00, 124.52 batch/s, accuracy=0.84, loss=0.412]
         Epoch 6: 100%|██████████| 340/340 [00:06<00:00, 50.09 batch/s, accuracy=1, loss=0.021]
    Validation: 6: 100%|██████████| 76/76 [00:00<00:00, 138.31 batch/s, accuracy=0.9, loss=0.227]
         Epoch 7: 100%|██████████| 340/340 [00:06<00:00, 49.07 batch/s, accuracy=1, loss=0.225]
    Validation: 7: 100%|██████████| 76/76 [00:00<00:00, 130.48 batch/s, accuracy=0.88, loss=0.282]
         Epoch 8: 100%|██████████| 340/340 [00:06<00:00, 49.27 batch/s, accuracy=1, loss=0.119]
    Validation: 8: 100%|██████████| 76/76 [00:00<00:00, 152.10 batch/s, accuracy=0.86, loss=0.351]
         Epoch 9: 100%|██████████| 340/340 [00:07<00:00, 48.11 batch/s, accuracy=1, loss=0.0589]
    Validation: 9: 100%|██████████| 76/76 [00:00<00:00, 127.25 batch/s, accuracy=0.88, loss=0.314]
        Epoch 10: 100%|██████████| 340/340 [00:06<00:00, 54.11 batch/s, accuracy=1, loss=0.0574]
   Validation: 10: 100%|██████████| 76/76 [00:00<00:00, 154.88 batch/s, accuracy=0.8, loss=0.403]
        Epoch 11: 100%|██████████| 340/340 [00:05<00:00, 56.80 batch/s, accuracy=1, loss=0.254]
   Validation: 11: 100%|██████████| 76/76 [00:00<00:00, 163.88 batch/s, accuracy=0.96, loss=0.216]
        Epoch 12: 100%|██████████| 340/340 [00:05<00:00, 57.47 batch/s, accuracy=1, loss=0.156]
   Validation: 12: 100%|██████████| 76/76 [00:00<00:00, 153.47 batch/s, accuracy=0.82, loss=0.487]
        Epoch 13: 100%|██████████| 340/340 [00:05<00:00, 57.83 batch/s, accuracy=1, loss=0.0658]
   Validation: 13: 100%|██████████| 76/76 [00:00<00:00, 154.07 batch/s, accuracy=0.92, loss=0.2]
        Epoch 14: 100%|██████████| 340/340 [00:05<00:00, 56.98 batch/s, accuracy=1, loss=0.0601]
   Validation: 14: 100%|██████████| 76/76 [00:00<00:00, 147.39 batch/s, accuracy=0.94, loss=0.237]
        Epoch 15: 100%|██████████| 340/340 [00:06<00:00, 53.63 batch/s, accuracy=1, loss=0.033]
   Validation: 15: 100%|██████████| 76/76 [00:00<00:00, 120.75 batch/s, accuracy=0.92, loss=0.215]
        Epoch 16: 100%|██████████| 340/340 [00:07<00:00, 47.01 batch/s, accuracy=1, loss=0.00694]
   Validation: 16: 100%|██████████| 76/76 [00:00<00:00, 138.89 batch/s, accuracy=0.86, loss=0.561]
        Epoch 17: 100%|██████████| 340/340 [00:06<00:00, 50.09 batch/s, accuracy=1, loss=0.0298]
   Validation: 17: 100%|██████████| 76/76 [00:00<00:00, 124.71 batch/s, accuracy=0.94, loss=0.0985]
        Epoch 18: 100%|██████████| 340/340 [00:06<00:00, 49.90 batch/s, accuracy=0.75, loss=0.295]
   Validation: 18: 100%|██████████| 76/76 [00:00<00:00, 139.97 batch/s, accuracy=0.86, loss=0.52]
        Epoch 19: 100%|██████████| 340/340 [00:06<00:00, 50.88 batch/s, accuracy=1, loss=0.033]
   Validation: 19: 100%|██████████| 76/76 [00:00<00:00, 116.70 batch/s, accuracy=0.88, loss=0.376]
        Epoch 20: 100%|██████████| 340/340 [00:06<00:00, 49.02 batch/s, accuracy=1, loss=0.0197]
   Validation: 20: 100%|██████████| 76/76 [00:00<00:00, 116.72 batch/s, accuracy=0.98, loss=0.0836]
        Epoch 21: 100%|██████████| 340/340 [00:06<00:00, 50.68 batch/s, accuracy=1, loss=0.075]
   Validation: 21: 100%|██████████| 76/76 [00:00<00:00, 131.27 batch/s, accuracy=0.9, loss=0.379]
        Epoch 22: 100%|██████████| 340/340 [00:06<00:00, 50.23 batch/s, accuracy=1, loss=0.0487]
   Validation: 22: 100%|██████████| 76/76 [00:00<00:00, 125.02 batch/s, accuracy=0.84, loss=0.369]
        Epoch 23: 100%|██████████| 340/340 [00:06<00:00, 51.22 batch/s, accuracy=1, loss=0.0527]
   Validation: 23: 100%|██████████| 76/76 [00:00<00:00, 136.93 batch/s, accuracy=0.94, loss=0.227]
        Epoch 24: 100%|██████████| 340/340 [00:06<00:00, 51.04 batch/s, accuracy=1, loss=0.0666]
   Validation: 24: 100%|██████████| 76/76 [00:00<00:00, 132.07 batch/s, accuracy=0.96, loss=0.583]
        Epoch 25: 100%|██████████| 340/340 [00:06<00:00, 51.19 batch/s, accuracy=1, loss=0.0122]
   Validation: 25: 100%|██████████| 76/76 [00:00<00:00, 138.55 batch/s, accuracy=0.94, loss=0.159]
        Epoch 26: 100%|██████████| 340/340 [00:06<00:00, 48.99 batch/s, accuracy=1, loss=0.0256]
   Validation: 26: 100%|██████████| 76/76 [00:00<00:00, 146.68 batch/s, accuracy=0.86, loss=0.325]
        Epoch 27: 100%|██████████| 340/340 [00:06<00:00, 51.50 batch/s, accuracy=1, loss=0.0169]
   Validation: 27: 100%|██████████| 76/76 [00:00<00:00, 137.55 batch/s, accuracy=0.88, loss=0.251]
        Epoch 28: 100%|██████████| 340/340 [00:06<00:00, 50.05 batch/s, accuracy=0.75, loss=1.6]
   Validation: 28: 100%|██████████| 76/76 [00:00<00:00, 136.20 batch/s, accuracy=0.82, loss=0.464]
        Epoch 29: 100%|██████████| 340/340 [00:06<00:00, 49.98 batch/s, accuracy=1, loss=0.0767]
   Validation: 29: 100%|██████████| 76/76 [00:00<00:00, 123.22 batch/s, accuracy=0.84, loss=0.693]
        Epoch 30: 100%|██████████| 340/340 [00:07<00:00, 48.35 batch/s, accuracy=1, loss=0.0422]
   Validation: 30: 100%|██████████| 76/76 [00:00<00:00, 132.65 batch/s, accuracy=0.96, loss=0.215]
        Epoch 31: 100%|██████████| 340/340 [00:06<00:00, 50.11 batch/s, accuracy=1, loss=0.0102]
   Validation: 31: 100%|██████████| 76/76 [00:00<00:00, 134.87 batch/s, accuracy=0.9, loss=0.206]
        Epoch 32: 100%|██████████| 340/340 [00:06<00:00, 50.93 batch/s, accuracy=1, loss=0.154]
   Validation: 32: 100%|██████████| 76/76 [00:00<00:00, 142.81 batch/s, accuracy=0.94, loss=0.318]
        Epoch 33: 100%|██████████| 340/340 [00:06<00:00, 50.37 batch/s, accuracy=1, loss=0.0167]
   Validation: 33: 100%|██████████| 76/76 [00:00<00:00, 133.03 batch/s, accuracy=0.9, loss=0.421]
        Epoch 34: 100%|██████████| 340/340 [00:06<00:00, 50.19 batch/s, accuracy=1, loss=0.0273]
   Validation: 34: 100%|██████████| 76/76 [00:00<00:00, 139.28 batch/s, accuracy=0.92, loss=0.194]
        Epoch 35: 100%|██████████| 340/340 [00:06<00:00, 50.29 batch/s, accuracy=1, loss=0.00164]
   Validation: 35: 100%|██████████| 76/76 [00:00<00:00, 128.29 batch/s, accuracy=0.9, loss=0.422]
        Epoch 36: 100%|██████████| 340/340 [00:06<00:00, 49.49 batch/s, accuracy=0.75, loss=0.474]
   Validation: 36: 100%|██████████| 76/76 [00:00<00:00, 140.86 batch/s, accuracy=0.9, loss=0.539]
        Epoch 37: 100%|██████████| 340/340 [00:06<00:00, 50.29 batch/s, accuracy=1, loss=0.125]
   Validation: 37: 100%|██████████| 76/76 [00:00<00:00, 142.97 batch/s, accuracy=0.9, loss=0.23]
        Epoch 38: 100%|██████████| 340/340 [00:06<00:00, 51.02 batch/s, accuracy=1, loss=0.0303]
   Validation: 38: 100%|██████████| 76/76 [00:00<00:00, 125.85 batch/s, accuracy=0.96, loss=0.105]
        Epoch 39: 100%|██████████| 340/340 [00:06<00:00, 49.23 batch/s, accuracy=1, loss=0.0738]
   Validation: 39: 100%|██████████| 76/76 [00:00<00:00, 130.64 batch/s, accuracy=0.92, loss=0.452]
        Epoch 40: 100%|██████████| 340/340 [00:06<00:00, 50.23 batch/s, accuracy=1, loss=0.0343]
   Validation: 40: 100%|██████████| 76/76 [00:00<00:00, 126.79 batch/s, accuracy=0.9, loss=0.202]
        Epoch 41: 100%|██████████| 340/340 [00:06<00:00, 49.61 batch/s, accuracy=1, loss=0.0327]
   Validation: 41: 100%|██████████| 76/76 [00:00<00:00, 124.90 batch/s, accuracy=0.96, loss=0.18]
        Epoch 42: 100%|██████████| 340/340 [00:06<00:00, 49.66 batch/s, accuracy=1, loss=0.0553]
   Validation: 42: 100%|██████████| 76/76 [00:00<00:00, 138.55 batch/s, accuracy=0.94, loss=0.202]
        Epoch 43: 100%|██████████| 340/340 [00:06<00:00, 49.40 batch/s, accuracy=1, loss=0.022]
   Validation: 43: 100%|██████████| 76/76 [00:00<00:00, 144.19 batch/s, accuracy=0.88, loss=0.432]
        Epoch 44: 100%|██████████| 340/340 [00:06<00:00, 51.24 batch/s, accuracy=1, loss=0.0275]
   Validation: 44: 100%|██████████| 76/76 [00:00<00:00, 118.50 batch/s, accuracy=0.96, loss=0.157]
        Epoch 45: 100%|██████████| 340/340 [00:07<00:00, 48.55 batch/s, accuracy=1, loss=0.0225]
   Validation: 45: 100%|██████████| 76/76 [00:00<00:00, 117.10 batch/s, accuracy=0.96, loss=0.0771]
        Epoch 46: 100%|██████████| 340/340 [00:06<00:00, 49.09 batch/s, accuracy=1, loss=0.0735]
   Validation: 46: 100%|██████████| 76/76 [00:00<00:00, 137.30 batch/s, accuracy=0.96, loss=0.105]
        Epoch 47: 100%|██████████| 340/340 [00:07<00:00, 47.97 batch/s, accuracy=1, loss=0.0176]
   Validation: 47: 100%|██████████| 76/76 [00:00<00:00, 133.75 batch/s, accuracy=0.88, loss=0.795]
        Epoch 48: 100%|██████████| 340/340 [00:07<00:00, 48.42 batch/s, accuracy=1, loss=0.00247]
   Validation: 48: 100%|██████████| 76/76 [00:00<00:00, 121.33 batch/s, accuracy=0.96, loss=0.261]
        Epoch 49: 100%|██████████| 340/340 [00:07<00:00, 47.34 batch/s, accuracy=1, loss=0.00522]
   Validation: 49: 100%|██████████| 76/76 [00:00<00:00, 127.11 batch/s, accuracy=0.92, loss=0.217]
```

Generate Metrics on the Dev Set

Here we recreate the `dev` dataloader to go a single document at a time.

We use the less robust `predict` method so we can explicitly show the values and calculations being performed.

In [6]:
```
dev_dataloader = datasets.create_torch_dataloader(val_dataset, vocab,  label_transform, text_transform, weighte

model.eval()
preds = []
labels = []
for batch in dev_dataloader:
    label, text, *_ = batch
    pred_label = train.predict(model, text)
    preds.append(pred_label)
    labels.append(label.cpu().item())
```

Generate a Confusion Matrix

We can use the confusion matrix to make it easy to visualize the values for the metrics.

	P0	P1
A0	TN	FP
A1	FN	TP

In [7]:
```
from sklearn.metrics import confusion_matrix, average_precision_score
cm = confusion_matrix(labels, preds)
print(cm)
```

```
[[4352  348]
 [  94   56]]
```

Calculate Precision, Recall, F1-Score on Dev Set

We use a confusion matrix to make it easy to map out the values for true positive, true negative, false positive, and false negative.

- Precision = tp / (tp + fp)
- Recall = tp / (tp + fn)
- F1 Score = 2 *(precision* recall) / (precision + recall)

In [8]:
```
tn, fp, fn, tp = cm.ravel() # Extract the components

# Calculate and print Precision
precision_string = f"{tp} / ({tp} + {fp})"
precision = round(eval(precision_string), 4)
print(f"Precision: {precision_string} = {precision}")

# Calculate and print Recall
recall_string = f"{tp} / ({tp} + {fn})"
recall = round(eval(recall_string), 4)
print(f"Recall: {recall_string} = {recall}")

# Calculate and print F1 Score
f1_string = f"2 * ({precision} * {recall}) / ({precision} + {recall})"
f1 = eval(f1_string)
print(f"F1: {f1_string} = {round(f1, 4)}")

print(f"AP: {average_precision_score(labels, preds)}")
```

```
Precision: 56 / (56 + 348) = 0.1386
Recall: 56 / (56 + 94) = 0.3733
F1: 2 * (0.1386 * 0.3733) / (0.1386 + 0.3733) = 0.2021
AP: 0.07113061821646083
```

Update Methods

Because of the poor performance, we are going to update the vocab and model.

- Use GloVe embeddings within the model as well as the basis for the vocabulary.
- Update the model to be a 3-layer MLP with Activation to introduce non-linearity.

Repeat Experiment with "Title", "Abstract", and "Keyword" Data

We will keep everything exactly the same for setup, changing only the things needed.

In [9]:
```
use_columns = ["title", "abstract", "keywords"]
data_columns = [datafield_map[col] for col in use_columns]
ngrams = 1

glove = ir_vocab.create_glove_with_unk_vector()
vocab = ir_vocab.create_vocab_from_glove(glove)

# Load the TSV into datasets with the appropriate feature columns.
train_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic_review/phase1.train.shuf.tsv", data_colum
val_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic_review/phase1.dev.shuf.tsv", data_columns)

# Instantiate the dataloaders.
train_dataloader = datasets.create_torch_dataloader(train_dataset, vocab,  label_transform, text_transform, wei
val_dataloader = datasets.create_torch_dataloader(val_dataset, vocab,  label_transform, text_transform, weighte

# Create Model
model = models.EmbeddingBagMLPModel(num_class=num_classes, hidden_layer_size=100, embedding_vectors=glove.vecto
```

In [12]:
```
EPOCHS = 50
learning_rate = 0.001

# Create the loss function weighted to inverse class distribution
loss_function = torch.nn.CrossEntropyLoss()

# Instantiate a Stochastic Gradient Descent optimizer and "Auto" Learning Rate schedule.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.95)

# Tensorboard writing utility class.
writer = SummaryWriter("runs/TAK_MLP")

# Perform Training
for i in range(EPOCHS):
    start_iter = len(train_dataloader) * i
    train.train_epoch(i, model, optimizer, loss_function, train_dataloader, start_iter=start_iter, writer=write
    validation_results = train.evaluate_epoch(i, model, loss_function, val_dataloader, writer)
    scheduler.step()

torch.save(model.state_dict(), "model_weights/TAK_MLP_state_dict.pth")
```

```
         Epoch 0: 100%|██████████| 340/340 [00:14<00:00, 23.62 batch/s, accuracy=0.75, loss=0.994]
    Validation: 0: 100%|██████████| 76/76 [00:02<00:00, 29.22 batch/s, accuracy=0.78, loss=0.357]
         Epoch 1: 100%|██████████| 340/340 [00:13<00:00, 24.55 batch/s, accuracy=0.75, loss=0.299]
    Validation: 1: 100%|██████████| 76/76 [00:02<00:00, 29.60 batch/s, accuracy=0.9, loss=0.218]
         Epoch 2: 100%|██████████| 340/340 [00:13<00:00, 24.98 batch/s, accuracy=0.25, loss=1.29]
    Validation: 2: 100%|██████████| 76/76 [00:02<00:00, 28.30 batch/s, accuracy=0.76, loss=0.519]
         Epoch 3: 100%|██████████| 340/340 [00:13<00:00, 25.43 batch/s, accuracy=1, loss=0.179]
    Validation: 3: 100%|██████████| 76/76 [00:02<00:00, 25.76 batch/s, accuracy=0.84, loss=0.283]
         Epoch 4: 100%|██████████| 340/340 [00:13<00:00, 26.04 batch/s, accuracy=0.75, loss=0.538]
    Validation: 4: 100%|██████████| 76/76 [00:02<00:00, 29.29 batch/s, accuracy=0.84, loss=0.319]
         Epoch 5: 100%|██████████| 340/340 [00:12<00:00, 26.30 batch/s, accuracy=0.75, loss=0.337]
    Validation: 5: 100%|██████████| 76/76 [00:02<00:00, 29.53 batch/s, accuracy=0.78, loss=0.36]
         Epoch 6: 100%|██████████| 340/340 [00:12<00:00, 26.23 batch/s, accuracy=1, loss=0.0418]
    Validation: 6: 100%|██████████| 76/76 [00:02<00:00, 28.43 batch/s, accuracy=0.86, loss=0.319]
         Epoch 7: 100%|██████████| 340/340 [00:13<00:00, 25.90 batch/s, accuracy=1, loss=0.137]
    Validation: 7: 100%|██████████| 76/76 [00:02<00:00, 27.00 batch/s, accuracy=0.84, loss=0.294]
         Epoch 8: 100%|██████████| 340/340 [00:13<00:00, 25.74 batch/s, accuracy=0.75, loss=0.296]
    Validation: 8: 100%|██████████| 76/76 [00:02<00:00, 29.07 batch/s, accuracy=0.82, loss=0.328]
         Epoch 9: 100%|██████████| 340/340 [00:12<00:00, 26.52 batch/s, accuracy=0.75, loss=0.298]
    Validation: 9: 100%|██████████| 76/76 [00:02<00:00, 29.01 batch/s, accuracy=0.84, loss=0.322]
        Epoch 10: 100%|██████████| 340/340 [00:13<00:00, 25.97 batch/s, accuracy=1, loss=0.0966]
   Validation: 10: 100%|██████████| 76/76 [00:02<00:00, 27.86 batch/s, accuracy=0.74, loss=0.5]
        Epoch 11: 100%|██████████| 340/340 [00:13<00:00, 25.77 batch/s, accuracy=1, loss=0.0825]
   Validation: 11: 100%|██████████| 76/76 [00:02<00:00, 28.20 batch/s, accuracy=0.92, loss=0.196]
        Epoch 12: 100%|██████████| 340/340 [00:13<00:00, 25.55 batch/s, accuracy=0.75, loss=0.367]
   Validation: 12: 100%|██████████| 76/76 [00:02<00:00, 25.81 batch/s, accuracy=0.86, loss=0.205]
        Epoch 13: 100%|██████████| 340/340 [00:13<00:00, 25.51 batch/s, accuracy=1, loss=0.129]
   Validation: 13: 100%|██████████| 76/76 [00:02<00:00, 27.89 batch/s, accuracy=0.82, loss=0.349]
        Epoch 14: 100%|██████████| 340/340 [00:13<00:00, 25.17 batch/s, accuracy=1, loss=0.134]
   Validation: 14: 100%|██████████| 76/76 [00:02<00:00, 26.59 batch/s, accuracy=0.82, loss=0.279]
        Epoch 15: 100%|██████████| 340/340 [00:13<00:00, 25.09 batch/s, accuracy=0.75, loss=0.419]
   Validation: 15: 100%|██████████| 76/76 [00:03<00:00, 22.45 batch/s, accuracy=0.86, loss=0.304]
        Epoch 16: 100%|██████████| 340/340 [00:15<00:00, 22.66 batch/s, accuracy=1, loss=0.00779]
   Validation: 16: 100%|██████████| 76/76 [00:03<00:00, 22.72 batch/s, accuracy=0.82, loss=0.282]
        Epoch 17: 100%|██████████| 340/340 [00:15<00:00, 22.38 batch/s, accuracy=1, loss=0.125]
   Validation: 17: 100%|██████████| 76/76 [00:03<00:00, 23.23 batch/s, accuracy=0.8, loss=0.409]
        Epoch 18: 100%|██████████| 340/340 [00:15<00:00, 21.93 batch/s, accuracy=0.75, loss=0.758]
   Validation: 18: 100%|██████████| 76/76 [00:03<00:00, 24.47 batch/s, accuracy=0.82, loss=0.38]
        Epoch 19: 100%|██████████| 340/340 [00:15<00:00, 22.55 batch/s, accuracy=1, loss=0.168]
   Validation: 19: 100%|██████████| 76/76 [00:03<00:00, 24.59 batch/s, accuracy=0.88, loss=0.263]
        Epoch 20: 100%|██████████| 340/340 [00:14<00:00, 23.27 batch/s, accuracy=0.75, loss=0.746]
   Validation: 20: 100%|██████████| 76/76 [00:03<00:00, 24.57 batch/s, accuracy=0.82, loss=0.306]
        Epoch 21: 100%|██████████| 340/340 [00:14<00:00, 22.98 batch/s, accuracy=0.75, loss=0.327]
   Validation: 21: 100%|██████████| 76/76 [00:02<00:00, 26.71 batch/s, accuracy=0.94, loss=0.211]
        Epoch 22: 100%|██████████| 340/340 [00:13<00:00, 24.95 batch/s, accuracy=1, loss=0.0574]
   Validation: 22: 100%|██████████| 76/76 [00:02<00:00, 28.22 batch/s, accuracy=0.8, loss=0.344]
        Epoch 23: 100%|██████████| 340/340 [00:13<00:00, 25.97 batch/s, accuracy=1, loss=0.0406]
   Validation: 23: 100%|██████████| 76/76 [00:02<00:00, 25.36 batch/s, accuracy=0.88, loss=0.209]
        Epoch 24: 100%|██████████| 340/340 [00:14<00:00, 23.73 batch/s, accuracy=1, loss=0.0563]
   Validation: 24: 100%|██████████| 76/76 [00:02<00:00, 25.59 batch/s, accuracy=0.9, loss=0.143]
        Epoch 25: 100%|██████████| 340/340 [00:13<00:00, 25.84 batch/s, accuracy=0.75, loss=0.245]
   Validation: 25: 100%|██████████| 76/76 [00:02<00:00, 27.95 batch/s, accuracy=0.9, loss=0.157]
        Epoch 26: 100%|██████████| 340/340 [00:13<00:00, 25.51 batch/s, accuracy=1, loss=0.153]
   Validation: 26: 100%|██████████| 76/76 [00:02<00:00, 27.60 batch/s, accuracy=0.92, loss=0.188]
        Epoch 27: 100%|██████████| 340/340 [00:13<00:00, 25.56 batch/s, accuracy=1, loss=0.0932]
   Validation: 27: 100%|██████████| 76/76 [00:02<00:00, 27.51 batch/s, accuracy=0.96, loss=0.104]
        Epoch 28: 100%|██████████| 340/340 [00:13<00:00, 25.37 batch/s, accuracy=1, loss=0.0429]
   Validation: 28: 100%|██████████| 76/76 [00:02<00:00, 27.24 batch/s, accuracy=0.8, loss=0.439]
        Epoch 29: 100%|██████████| 340/340 [00:13<00:00, 25.26 batch/s, accuracy=1, loss=0.0793]
   Validation: 29: 100%|██████████| 76/76 [00:02<00:00, 28.76 batch/s, accuracy=0.88, loss=0.239]
        Epoch 30: 100%|██████████| 340/340 [00:13<00:00, 25.40 batch/s, accuracy=1, loss=0.117]
   Validation: 30: 100%|██████████| 76/76 [00:02<00:00, 28.14 batch/s, accuracy=0.92, loss=0.136]
        Epoch 31: 100%|██████████| 340/340 [00:13<00:00, 25.16 batch/s, accuracy=0.75, loss=0.479]
   Validation: 31: 100%|██████████| 76/76 [00:02<00:00, 28.87 batch/s, accuracy=0.9, loss=0.213]
        Epoch 32: 100%|██████████| 340/340 [00:13<00:00, 25.64 batch/s, accuracy=1, loss=0.038]
   Validation: 32: 100%|██████████| 76/76 [00:03<00:00, 22.90 batch/s, accuracy=0.88, loss=0.298]
        Epoch 33: 100%|██████████| 340/340 [00:14<00:00, 24.07 batch/s, accuracy=0.75, loss=0.246]
   Validation: 33: 100%|██████████| 76/76 [00:02<00:00, 27.83 batch/s, accuracy=0.82, loss=0.457]
        Epoch 34: 100%|██████████| 340/340 [00:13<00:00, 24.65 batch/s, accuracy=1, loss=0.0358]
   Validation: 34: 100%|██████████| 76/76 [00:03<00:00, 24.36 batch/s, accuracy=0.86, loss=0.226]
        Epoch 35: 100%|██████████| 340/340 [00:14<00:00, 24.02 batch/s, accuracy=1, loss=0.0942]
   Validation: 35: 100%|██████████| 76/76 [00:02<00:00, 27.15 batch/s, accuracy=0.9, loss=0.174]
        Epoch 36: 100%|██████████| 340/340 [00:13<00:00, 25.65 batch/s, accuracy=1, loss=0.0324]
   Validation: 36: 100%|██████████| 76/76 [00:02<00:00, 26.43 batch/s, accuracy=0.8, loss=0.427]
        Epoch 37: 100%|██████████| 340/340 [00:13<00:00, 25.20 batch/s, accuracy=1, loss=0.27]
   Validation: 37: 100%|██████████| 76/76 [00:02<00:00, 26.31 batch/s, accuracy=0.9, loss=0.165]
        Epoch 38: 100%|██████████| 340/340 [00:13<00:00, 25.65 batch/s, accuracy=1, loss=0.0189]
   Validation: 38: 100%|██████████| 76/76 [00:02<00:00, 27.20 batch/s, accuracy=0.9, loss=0.299]
        Epoch 39: 100%|██████████| 340/340 [00:13<00:00, 25.26 batch/s, accuracy=1, loss=0.0542]
   Validation: 39: 100%|██████████| 76/76 [00:02<00:00, 26.00 batch/s, accuracy=0.86, loss=0.213]
        Epoch 40: 100%|██████████| 340/340 [00:13<00:00, 25.22 batch/s, accuracy=1, loss=0.0553]
   Validation: 40: 100%|██████████| 76/76 [00:03<00:00, 25.12 batch/s, accuracy=0.78, loss=0.426]
        Epoch 41: 100%|██████████| 340/340 [00:13<00:00, 25.00 batch/s, accuracy=1, loss=0.064]
   Validation: 41: 100%|██████████| 76/76 [00:02<00:00, 26.98 batch/s, accuracy=0.9, loss=0.192]
        Epoch 42: 100%|██████████| 340/340 [00:13<00:00, 25.01 batch/s, accuracy=1, loss=0.0433]
   Validation: 42: 100%|██████████| 76/76 [00:02<00:00, 27.93 batch/s, accuracy=0.94, loss=0.127]
        Epoch 43: 100%|██████████| 340/340 [00:13<00:00, 24.92 batch/s, accuracy=0.75, loss=0.789]
   Validation: 43: 100%|██████████| 76/76 [00:02<00:00, 27.59 batch/s, accuracy=0.82, loss=0.32]
        Epoch 44: 100%|██████████| 340/340 [00:13<00:00, 25.21 batch/s, accuracy=0.75, loss=0.518]
   Validation: 44: 100%|██████████| 76/76 [00:02<00:00, 26.37 batch/s, accuracy=0.82, loss=0.291]
        Epoch 45: 100%|██████████| 340/340 [00:13<00:00, 24.77 batch/s, accuracy=0.75, loss=0.361]
   Validation: 45: 100%|██████████| 76/76 [00:02<00:00, 27.78 batch/s, accuracy=0.86, loss=0.279]
        Epoch 46: 100%|██████████| 340/340 [00:15<00:00, 22.41 batch/s, accuracy=1, loss=0.0211]
   Validation: 46: 100%|██████████| 76/76 [00:02<00:00, 26.13 batch/s, accuracy=0.92, loss=0.204]
        Epoch 47: 100%|██████████| 340/340 [00:14<00:00, 23.33 batch/s, accuracy=0.75, loss=0.401]
   Validation: 47: 100%|██████████| 76/76 [00:02<00:00, 25.71 batch/s, accuracy=0.96, loss=0.104]
        Epoch 48: 100%|██████████| 340/340 [00:13<00:00, 25.33 batch/s, accuracy=1, loss=0.0586]
   Validation: 48: 100%|██████████| 76/76 [00:03<00:00, 25.10 batch/s, accuracy=0.9, loss=0.194]
        Epoch 49: 100%|██████████| 340/340 [00:13<00:00, 25.73 batch/s, accuracy=1, loss=0.0563]
   Validation: 49: 100%|██████████| 76/76 [00:03<00:00, 20.82 batch/s, accuracy=0.88, loss=0.24]
```

In [13]:
```
dev_dataloader = datasets.create_torch_dataloader(val_dataset, vocab,  label_transform, text_transform, weighte

model.eval()
preds = []
labels = []
for batch in dev_dataloader:
    label, text, *_ = batch
    pred_label = train.predict(model, text)
    preds.append(pred_label)
    labels.append(label.cpu().item())

cm = confusion_matrix(labels, preds)
print(cm)
tn, fp, fn, tp = cm.ravel() # Extract the components

# Calculate and print Precision
precision_string = f"{tp} / ({tp} + {fp})"
precision = round(eval(precision_string), 4)
print(f"Precision: {precision_string} = {precision}")

# Calculate and print Recall
recall_string = f"{tp} / ({tp} + {fn})"
recall = round(eval(recall_string), 4)
print(f"Recall: {recall_string} = {recall}")

# Calculate and print F1 Score
f1_string = f"2 * ({precision} * {recall}) / ({precision} + {recall})"
f1 = eval(f1_string)
print(f"F1: {f1_string} = {round(f1, 4)}")


print(f"AP: {average_precision_score(labels, preds)}")
```

```
[[4120  580]
 [  24  126]]
Precision: 126 / (126 + 580) = 0.1785
Recall: 126 / (126 + 24) = 0.84
F1: 2 * (0.1785 * 0.84) / (0.1785 + 0.84) = 0.2944
AP: 0.15486346777255336
```

In [ ]:
```
%load_ext tensorboard
%tensorboard --logdir runs/
```

Generate Predictions On Test Data

In [29]:
```
test_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic_review/phase1.test.shuf.tsv", data_columns
test_dataloader = datasets.create_torch_dataloader(test_dataset, vocab, label_transform, text_transform,  weigh

model.eval()
preds = []
doc_ids = []
with open("RCERESA1.txt", "w") as f:
    for batch in test_dataloader:
        label, text, offsets, doc_id = batch
        pred_label = train.predict(model, text)

        pred = pred_label if pred_label == 1 else -1
        line = f"{doc_id[0]}\t{pred}\n"
        f.write(line)
```

Custom Modules - Source Code

A number of custom code was generated to support this experiment.

datasets.py

```python
import io
from typing import Callable, List

import torch
from torch.utils import data
from torch.utils.data.sampler import WeightedRandomSampler
from torchtext.data.utils import get_tokenizer
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import Vocab

_default_tokenizer = get_tokenizer("basic_english")
DEFAULT_LABEL_TRANSFORM = lambda x: x
DEFAULT_TEXT_TRANSFORM = lambda x: _default_tokenizer(x)


def create_torch_dataloader(
    dataset: data.Dataset,
    vocab: Vocab,
    label_transform: Callable = DEFAULT_LABEL_TRANSFORM,
    text_transform: Callable = DEFAULT_TEXT_TRANSFORM,
    weighted=True,
    **kwargs
):
    """Creates a Pytorch style dataloader using a dataset and a precompiled vocab.

    The dataset returns "model-ready" data.

    Args:
        dataset: The raw text dataset to be used during inference
        vocab: the premade vocabulary used to index words/phrases
        label_transform: any operation used on the datasets label output
        text_transform: operation used on the raw text sentence outputs from the data
        weighted: whether to weight the samples based on class distribution
        **kwargs: any additional kwargs used by Pytorch DataLoaders.

    Returns:
        A PyTorch DataLoader to be used during training, eval, or test.
    """
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def _collate_batch(batch):
        label_list, docid_list, text_list, offsets = [], [], [], [0]
        for (_label, _docid, _text) in batch:
            label_list.append(label_transform(_label))
            processed_text = torch.tensor(
                vocab(text_transform(_text)), dtype=torch.int64
            )
            text_list.append(processed_text)
            offsets.append(processed_text.size(0))
            docid_list.append(_docid)
        label_list = torch.tensor(label_list, dtype=torch.int64)
        offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
        text_list = torch.cat(text_list)
        return label_list.to(device), text_list.to(device), offsets.to(device), docid_list

    if weighted:
        weights = dataset.sample_weights
        sampler = WeightedRandomSampler(weights=weights, num_samples=len(weights))
    else:
        sampler = None

    return data.DataLoader(
        dataset,
        collate_fn=_collate_batch,
        shuffle=(sampler is None),
        sampler=sampler,
        **kwargs
    )


class TSVRawTextIterableDataset(data.IterableDataset):
    """Dataset that loads TSV data incrementally as an iterable and returns raw text.

    This dataset must be traversed in order as it only reads from the TSV file as it is called.
    Useful if the size of data is too large to load into memory at once.
    """

    def __init__(self, filepath: str, data_columns: List[int]):
        """Loads an iterator from a file.

        Args:
            filepath: location of the .tsv file
            data_columns: the columns in the .tsv that are used as feature data
        """
        self._number_of_items = _get_tsv_file_length(filepath)
        self._iterator = _create_data_from_tsv(
            filepath, data_column_indices=data_columns
        )
        self._current_position = 0

    def __iter__(self):
        return self

    def __next__(self):
        item = next(self._iterator)
        self._current_position += 1
        return item

    def __len__(self):
        return self._number_of_items


class TSVRawTextMapDataset(data.Dataset):
    """Dataset that loads all TSV data into memory and returns raw text.

    This dataset provides a map interface, allowing access to any entry.
    Useful for modifying the sampling or order during training.
    """

    def __init__(self, filepath: str, data_columns: List[int]):
        """Loads .tsv structed data into memory.

        Args:
            filepath: location of the .tsv file
            data_columns: the columns in the .tsv that are used as feature data
        """
        self._records = list(
            _create_data_from_tsv(filepath, data_column_indices=data_columns)
        )
        self._sample_weights, self._class_weights = self._calculate_weights()

    @property
    def sample_weights(self):
        return self._sample_weights

    @property
    def class_weights(self):
        return self._class_weights

    def _calculate_weights(self):
        targets = torch.tensor(
            [label if label > 0 else 0 for label, *_ in self._records]
        )
        unique, sample_counts = torch.unique(targets, return_counts=True)
        weight = 1.0 / sample_counts
        sample_weights =  torch.tensor([weight[t] for t in targets])
        class_weights = weight / weight.sum()
        return sample_weights, class_weights

    def __getitem__(self, index):
        return self._records[index]

    def __len__(self):
        return len(self._records)


def _create_data_from_tsv(data_path, data_column_indices):
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\t")
        for row in reader:
            data = [row[i] for i in data_column_indices]
            yield int(row[0]), row[1], " ".join(data)


def _get_tsv_file_length(data_path):
    with io.open(data_path, encoding="utf8") as f:
        row_count = sum(1 for row in f)

    return row_count
```

models.py

```python
import io
from typing import List
import numpy as np
import torch
from torchtext.data.utils import get_tokenizer, ngrams_iterator
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import build_vocab_from_iterator
from torchtext.vocab import vocab as vocab_builder
from torchtext.vocab import GloVe


def create_glove_with_unk_vector() -> GloVe:
    glove = GloVe()
    # Load the average vector for this glove embedding set to use for defaults.
    average_glove_vector = np.load("../datasets/glove_default_vector.npy")
    unk_init_vec = torch.from_numpy(average_glove_vector)
    # Extend the glove vectors with one for "unk"
    glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))

    return glove

def create_vocab_from_glove(glove: GloVe):
    # Since glove is already ordered and not a counter, we overload the
    # Constructor to align the indices.
    unk_token = "<unk>"
    vocab = vocab_builder(glove.stoi, min_freq=0)
    vocab.append_token(unk_token)
    vocab.set_default_index(vocab[unk_token])

    return vocab

def create_vocab_from_tsv(
    filepath: str,
    column_indices_to_use: List[int],
    minimum_word_freq: int = 1,
    ngrams: int = 1,
):
    """Creates a PyTorch vocab object from a TSV file.

    The resulting vocab object converts words to indices for assisting in embedding and DL operations.

    Args:
        filepath: The location of the TSV file
        minimum_word_freq: How many times a word must appear to be included
        ngrams: The size of ngrams to use for the vocab
        column_indices_to_use: Which columns from the TSV are part of the actual feature set

    Returns:
        A torchtext vocab object.
    """
    unk_token = "<unk>"
    vocab = build_vocab_from_iterator(
        _tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use),
        min_freq=minimum_word_freq,
        specials=[unk_token],
    )
    vocab.set_default_index(vocab[unk_token])
    return vocab


def _tsv_iterator(data_path, ngrams, column_indices):
    # Spacy has novel tokenizer
    tokenizer = get_tokenizer("basic_english")
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\t")
        for row in reader:
            row_iter = [row[i] for i in column_indices]
            tokens = " ".join(row_iter)
            yield ngrams_iterator(tokenizer(tokens), ngrams)
```

train.py

```python
from collections import Counter
from logging import log
from typing import Any, Callable, Dict, Tuple

import torch
import torch.nn as nn
from sklearn.metrics import precision_recall_fscore_support, average_precision_score
from torch.utils import data
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm


def predict(model: nn.Module, text: torch.Tensor) -> int:
    """Predicts the class of a specific text given converted features.

    Args:
        model: the model to use for prediction/inferrence
        text: the previously converted text (using the prior dictionary)

    Returns:
        The predicted label for the provided text.
    """
    model.eval()
    no_offset = torch.tensor([0])
    with torch.no_grad():
        pred_scores = model(text, no_offset)
        pred_label = pred_scores.argmax(1).item()
        return pred_label


def train_epoch(
    epoch_num: int,
    model: nn.Module,
    optimizer: torch.optim.Optimizer,
    loss_function: Callable,
    dataloader: data.DataLoader,
    start_iter: int = 0,
    log_interval: int = 100,
    writer: SummaryWriter = None,
) -> int:
    """Performs training on a single pass through a dataloader.

    Args:
        epoch_num: The current number of this epoch of training.
        model: The PyTorch module to train
        optimizer: The optimizer to use for training.
        loss_function: The function that calculates loss between truth and prediction.
        dataloader: Provides a properly formatted batch of data at each iteration.
        start_iter: The iteration this epoch started on. Used for plotting.
        log_interval: How often to log the scores.
        writier: Tensorboard summary writer.

    Returns:
        The value of the start_iter plus number of batches performed this epoch.
    """
    batch_counter = start_iter
    model.train()
    with tqdm(dataloader, unit=" batch", bar_format="{desc:>20}{percentage:3.0f}%|{bar}{r_bar}") as tepoch:
        for batch in tepoch:
            batch_counter += 1
            tepoch.set_description(f"Epoch {epoch_num}")
            results = train_step(batch, model, optimizer, loss_function)
            tepoch.set_postfix(loss=results["loss"], accurracy=results["accuracy"])

            if writer is not None and batch_counter % log_interval == 0:
                writer.add_scalars("training", results, batch_counter)

    return batch_counter


def train_step(
    batch: Tuple[torch.Tensor, ...],
    model: nn.Module,
    optimizer: torch.optim.Optimizer,
    loss_function: Callable,
) -> Dict[str, float]:
    """Performs a single training step on a model.

    Args:
        batch: A previously formatted batch of data.
        model: Torch model to perform training on.
        optimizer: The optmizer class used in training
        loss_function: The callable function to generate loss between prediction and truth.

    Returns:
        The different metrics generated this training step.
    """
    labels, text, offsets, *_ = batch

    optimizer.zero_grad()
    predicted_scores = model(text, offsets)
    loss = loss_function(predicted_scores, labels)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
    optimizer.step()

    predicted_labels = predicted_scores.argmax(1)

    accuracy = (predicted_labels == labels).sum().item() / labels.size(0)

    y_true = labels.detach().cpu().numpy()
    y_pred = predicted_labels.cpu().numpy()
    precision, recall, fscore, support = precision_recall_fscore_support(
        y_true,
        y_pred,
        average="binary",
        zero_division=0,
    )

    results = {
        "loss": loss.detach().cpu().item(),
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "fscore": fscore,
    }
    return results


def evaluate_epoch(
    epoch_num: int,
    model: nn.Module,
    loss_function: Callable,
    dataloader: data.DataLoader,
    writer: SummaryWriter = None,
)-> Dict[str, float]:
    """Performs validation on a single pass through a dataloader.

    Args:
        epoch_num: The current number of this epoch of training.
        model: The PyTorch module to train
        loss_function: The function that calculates loss between truth and prediction.
        dataloader: Provides a properly formatted batch of data at each iteration.
        writier: Tensorboard summary writer.

    Returns:
        The average validation metrics for the whole dataset.
    """
    model.eval()
    aggregate_results = Counter()
    with tqdm(dataloader, unit=" batch", bar_format="{desc:>20}{percentage:3.0f}%|{bar}{r_bar}") as tepoch:
        for batch in tepoch:
            tepoch.set_description(f"Validation: {epoch_num}")
            results = evaluate_step(batch, model, loss_function)
            tepoch.set_postfix(loss=results["loss"], accurracy=results["accuracy"])
            aggregate_results += Counter(results)

        average_results = {
            key: aggregate_results[key] / tepoch.total for key in aggregate_results
        }

    writer.add_scalars("validation", average_results, epoch_num)
    return average_results


def evaluate_step(
    batch: Tuple[torch.Tensor, ...],
    model: nn.Module,
    loss_function: Callable,
) -> Dict[str, float]:
    """Performs a single validation step on a model.

    Args:
        batch: A previously formatted batch of data.
        model: Torch model to perform training on.
        loss_function: The callable function to generate loss between prediction and truth.

    Returns:
        The different metrics generated this training step.
    """

    labels, text, offsets, *_ = batch

    with torch.no_grad():
        predicted_scores = model(text, offsets)
        loss = loss_function(predicted_scores, labels)
        predicted_labels = predicted_scores.argmax(1)
        accuracy = (predicted_labels == labels).sum().item() / labels.size(0)

        y_true = labels.detach().cpu().numpy()
        y_pred = predicted_labels.cpu().numpy()

        precision, recall, fscore, support = precision_recall_fscore_support(
            y_true,
            y_pred,
            average="binary",
            zero_division=0,
        )

    results = {
        "loss": loss.detach().cpu().item(),
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "fscore": fscore,
    }
    return results
```

vocab.py

```python
import io
from typing import List
import numpy as np
import torch
from torchtext.data.utils import get_tokenizer, ngrams_iterator
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import build_vocab_from_iterator
from torchtext.vocab import vocab as vocab_builder
from torchtext.vocab import GloVe


def create_glove_with_unk_vector() -> GloVe:
    glove = GloVe()
    # Load the average vector for this glove embedding set to use for defaults.
    average_glove_vector = np.load("../datasets/glove_default_vector.npy")
    unk_init_vec = torch.from_numpy(average_glove_vector)
    # Extend the glove vectors with one for "unk"
    glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))

    return glove

def create_vocab_from_glove(glove: GloVe):
    # Since glove is already ordered and not a counter, we overload the
    # Constructor to align the indices.
    unk_token = "<unk>"
    vocab = vocab_builder(glove.stoi, min_freq=0)
    vocab.append_token(unk_token)
    vocab.set_default_index(vocab[unk_token])

    return vocab

def create_vocab_from_tsv(
    filepath: str,
    column_indices_to_use: List[int],
    minimum_word_freq: int = 1,
    ngrams: int = 1,
):
    """Creates a PyTorch vocab object from a TSV file.

    The resulting vocab object converts words to indices for assisting in embedding and DL operations.

    Args:
        filepath: The location of the TSV file
        minimum_word_freq: How many times a word must appear to be included
        ngrams: The size of ngrams to use for the vocab
        column_indices_to_use: Which columns from the TSV are part of the actual feature set

    Returns:
        A torchtext vocab object.
    """
    unk_token = "<unk>"
    vocab = build_vocab_from_iterator(
        _tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use),
        min_freq=minimum_word_freq,
        specials=[unk_token],
    )
    vocab.set_default_index(vocab[unk_token])
    return vocab


def _tsv_iterator(data_path, ngrams, column_indices):
    # Spacy has novel tokenizer
    tokenizer = get_tokenizer("basic_english")
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\t")
        for row in reader:
            row_iter = [row[i] for i in column_indices]
            tokens = " ".join(row_iter)
            yield ngrams_iterator(tokenizer(tokens), ngrams)
```